IPNAT Processing

More efficient searches

Thu, May 16, 2002

IPNAT table searching

A nonvolatile memory table that is searched often is the Node Address Table, or IPNAT, whose entries hold an IP address and a node number. When a server request is active and replies are received, and the source IP address is known, it is necessary to determine what node number is associated with this IP address. That is done by search for a match on the IP address in this table. (The IP address can also be found by a similar search of this table when the node number is known.) For PowerPC nodes, this table is in nonvolatile memory, so the search time suffers from slow access times. How can this be speeded up?

Dynamic memory copy scheme

For the IPNAT, there is much that must be retained across resets. Suppose we try to copy the table contents at reset time into another area of dynamic memory, to be used for searching.

The IPNAT is not static, however. Every second, all entries are examined, and two one-byte counters are used in each entry for counting out time when the Domain Name Service is consulted to refresh the IP address of that entry. The logic that supports all this is in the `LOOPDNSQ` local application. Let us assume that this logic can just as well operate on a dynamic memory copy of the IPNAT.

When a new entry is installed in the dynamic copy, it must also be installed in the nonvolatile copy. This can be done by `LOOPDNSQ` when it receives a reply message from the DNS. If it was a new node, it can place the result into the nonvolatile copy. If it was any change in the IP address of the dynamic entry, it should also update the nonvolatile copy. In this way, a surprising reset of the node should not lose any IPNAT information.

When coming up from reset, and thus creating the IPNAT copy, it is necessary to establish the counter values that should exist in each entry, since these will not change during normal operation. Indeed, the only information retained across resets would be whether the counter is nonzero. In the case that it is zero, it means that the DNS is not to be consulted, since the entry must be a foreign node for which the correct suffix is unknown. (Only one suffix is used in DNS queries, usually "`fnal.gov`," which is part of the IPNAT header. All local node numbers must be registered in the DNS as "`nodexxxx`," in which `xxxx` is the hexadecimal representation of the two-byte node number, all of which are in the range `0x05zz`—`0x07zz`, where `zz` is in the range `0x00`—`0xEF`.) The counters may be set so that the DNS is refreshed for each entry soon after the system is initialized. The `LOOPDNSQ` logic insures that no more than one DNS query will be made per second so as not to unduly burden the DNS server.

It would be useful to find all the places in the code where IPNAT is accessed. Since the IPNAT will still be housed in nonvolatile memory, according to its table directory entry, each user of it needs to be modified so that the copy in dynamic memory is used.

Consideration might be given to the idea of combining the nonvolatile information in the IPNAT with the IP security table information in the IPARP table. From this information, two dynamic memory copies could be initialized from a single official nonvolatile memory table.

Uses of IP-related searches in system code

There are four routines that support searches of IP-related network tables. One is PSNIPARP, whose job it is to return a pseudo node#, given an IP address and a UDP port#. The pseudo node# is then used as a 16-bit reference to a socket. Its meaning relates to an active entry in the IPARP table, which is an ARP cache plus a record of active ports. Up to 250 active entries are possible in IPARP, and each one can have up to 15 active ports at once.

The other three search routines are the IPNodeN, GtNodeN, and FindUDP routines. Both IPNodeN and GtNodeN search the IPNAT, whose entries hold a native node# and an IP address. The DNSQ local application is used to maintain the contents of this cache via periodic queries made to the Domain Name Service. This works because each native node# is registered with the local DNS with the name "nodexxxx.fnal.gov", where xxxx is the hexadecimal representation of the native node#. (For uses outside Fermilab, it may be configured to use a different suffix.)

IPNodeN uses a native node# to find an IP address via a search of IPNAT, then calls PSNIPARP to return a pseudo node#. GtNodeN uses an IP address to obtain a native node# via a search of IPNAT.

FindUDP uses pseudo node# to get an IP address to obtain Acnet node# via TRUNK search.

ACReq and Classic tasks call GtNodeN. It is necessary to have the native node# for each node returning a reply message, so that the data in that reply can be copied into the appropriate buffer. Associated with each active request structure is a list of native node#s that were found in the idents specified in the request. This logic applies only for server-type requests, both Classic and Acnet, as well as locally-initiated Classic requests.

OUTPQX calls IPNodeN to obtain a pseudo node# given a native node#. This is needed when a target node for a request message is in the 0x0500—0x07FF native node# range. It doesn't apply to replies, which would already have a pseudo-node# target.

OUTPQX calls FindUDP to obtain an Acnet node# when the target node# is a pseudo-node#, and the message to be queued to the network is an Acnet message.

During system initialization, InzSys calls InzIPNAT. This initializes the IPNAT header, but many of the fields are initialized by the DNSQ local application, which sorts the entries already in the table and manages any needed communications with the DNS.

Table lookup scheme

How much of this work can be supported via table lookup instead of searches? Recognizing that Fermilab uses Class B internet addresses, one can imagine a table indexed by Fermilab Class B addresses that would contain 64K entries, each of which could house a native node# and an Acnet node#, thus requiring 256K bytes of dynamic memory. This table would be populated at reset time by reviewing the contents of the nonvolatile IPNAT and TRUNK tables. The contents of this table would have to be maintained as new changes are received from occasional queries of the TRUNK tables sent by the AAUX local application, or from replies to DNS requests sent out by the DNSQ local application. Armed with such a table, searches of IPNAT for a match on an IP address, or searches of TRUNK for a match on an IP address, would be unnecessary. Each of these searches would be replaced by a table lookup.

This would speed up greatly the routines IPNodeN, GtNodeN, and FindUDP. (For unusual cases in which the IP address sought is not a Fermilab address, the old scheme could be used.)

IPNodeN searches the IPNAT for a match on a native node# in order to get the IP address, then converts it to a pseudo node#. This call is made from OUTPQX. But this search could be speeded up by a smaller table lookup. The range of possible native node#s is only 0x0300. A table of this many 4-byte entries could hold the IP addresses sought, requiring only 3K bytes. Again, it would have to be updated as new replies are received by DNSQ from the DNS. To make this table more general, one may expand it to 128K bytes, so that any node# in the range 0–0x7FFF would be a possible index to access the relevant IP address. In that case, it should be updated by AAUX replies from TRUNK table access, too, as well as whenever an IPARP table entry is established or cleared.

IPARP searching

Can PSNIPARP searching for a match on an IP address be changed to a table lookup? Perhaps there can be a larger entry size for the table indexed by Class B addresses, so there would be room for a pseudo-node# base, such as 0x6070. The port# block, with its list of active port#s, would still have to be accessed to determine the low 4-bits. If 8-byte entries were used, requiring 512K bytes, then one more 2-byte word would be available to house something else as yet to be determined. (8-byte entries seem nicer than 6-byte entries.)

Lookup table considerations

Call the new lookup table IPNOD. For IRMs, it can be located at 0x280000–0x2FFFFF. The only use so far made of the 0x200000 megabyte is as a temporary buffer for copying system code into onboard flash memory in node0562. But the system code is less than 128K bytes in size, so only 0x200000–0x21FFFF is needed for that. An entry in IPNOD will use 8 bytes, structured as

Field	*Size*	*Meaning*
nNode	2	Native node# (range 0x0500–0x07EF)
aNode	2	Acnet node# (range 0x0900–0x10EF)
pNode	2	Pseudo node# base (range 0x6020–0x6FF0)
sNode	2	(spare)

When an entry is placed into IPARP by the PSNIPARP routine, the corresponding entry in IPNOD must also be updated. When an Acnet node# is captured and installed via NodIPARP, a check should be made in this table, too. If the Acnet node# entry does not match, it would be useful to have a record that this happened.

When AAUX is about to update an entry in the TRUNK table, it should also update the corresponding IPNOD entry. It might be useful to have a record of whenever this changes from one nonzero value to another. (It is not necessary to record when an update replaces an aNode field that has a zero value.)

When DNSQ receives a reply, the corresponding IPNOD table entry should be updated. If there is a change, a record of the change should be made.

When the system resets, how shall the IPNOD table be initialized? For each IPNAT entry, the IP address is used to specify an IPNOD entry whose nNode field can be set. Similarly, for

each TRUNK table entry, the IP address is used to specify an IPNOD entry whose `aNode` field can be set. The IPARP table is empty, so nothing can be done at initialization time to set any of the pNode fields of an IPNOD entry.

A set of routines is needed to support the new IPNOD table.

```
Procedure InzIPNOD;

Function GetNNode(ipa: Longint): Integer;
Function GetANode(ipa: Longint): Integer;
Function GetPNode(ipa: Longint): Integer;
Function GetSNode(ipa: Longint): Integer;

Function SetNNode(ipa: Longint; newVal: Integer): Integer;
Function SetANode(ipa: Longint; newVal: Integer): Integer;
Function SetPNode(ipa: Longint; newVal: Integer): Integer;
Function SetSNode(ipa: Longint; newVal: Integer): Integer;

Function GetIPADD(node: Integer): Longint;
Procedure MonIPNOD;
```

The first routine initializes both the IPNOD and IPADD tables. (More about IPADD later.)

The first four Get routines access individual 2-byte fields in the entry indexed by an IP address. If one of these routines is called with an IP address argument that is outisde the assumed local Class B internet range, the return value is –1. The four Set routines return a nonzero value that is the old value, only in the case that the old value does not match the nonzero new value. A record is automatically written into a data stream for such cases. The record written includes the IP address, the new value, the old value, and the date/time. Note that the rotines `GetSNode` and `SetSNode` have no use yet, but they perform similar duties with the sNode field of the IPNOD entry.

When `TimIPARP` clears an IPARP entry, due to a timeout from inactivity, the `pNode` field is cleared in the corresponding IPNOD entry via a call to `SetPNode`.

The `GetIPADD` routine uses a table lookup based on any node# via the IPADD table of IP addresses. Note that if a pseudo node# is used, it must be the pseudo node# base; *i.e.*, the low 4 bits must be zero.

Changes in code required:

`PsnIPARP` in `IPARP.a`

When creating a new entry in IPARP, install base pseudo node# via `SetPNode`. When searching for a match in IPARP for a given IP address, call `GetPNode`. If the returned value is nonzero, use it to determine matching entry rather than searching IPARP.

`NodIPARP` in `IPARP.a`

If node# in Acnet range, call `SetANode` to install in IPNOD table.

`TimIPARP` in `IPARP.a`

When calling `ZIPARP` to clear an IPARP table entry, also call `SetPNode` to clear the `pNode` field in the IPNOD entry, as the pseudo node# is no longer valid.

`GtNodeN` in `IPNodeN.a`

Call `GetNNode` to get native node#. If this fails, there is no reason to search IPNAT, since it will not be found. But in case that an entry is made manually, it may be worth a try.

`FindUDP` in `IPNodeN.a`

At first, use the pseudo node# argument to get an IP address. Call `GetANode` to avoid searching TRUNK table. If this fails, there may still be a need to search TRUNK, as it may be that the IP address is not in the local Class B range.

`InzIPNAT` in `InzSys.a`

Completely initialize IPNOD table as well as initialize part of the IPNAT header. the first step is to clear all 512K of memory used for IPNOD. Then scan through each IPNAT entry and install each one into the IPNOD table via `SetNNode`. It should also review the TRUNK table and install each entry it finds there via `SetANode`. With these two scans, the IPNOD table should be "open for business."

`MonIPNOD` in `IPNODGS.a`

Monitor the entries placed into the `IPNODLOG` data stream and process them. More will appear about this later.

`LOOPDNSQ`

When receiving a valid reply from the DNS, make sure the information is also placed into IPNOD table via a call to `SetNNode`.

`LOOPAAUX`

When receiving a new TRUNK table, call `SetANode` for each entry to install them into IPNOD. This also includes receiving an unsolicited update message that carries such new TRUNK data. If a nonzero value should be returned, it means that there has been a change in the Acnet node# assigned to that IP address. This will automatically be logged.

`IPNodeN`

The second lookup table IPADD helps to support the mapping of a node# to an IP address, after which a call to `PsnIPARP` obtains the pseudo node#.

The ideas expressed here are developed further in the note "Table Lookup for IP Addresses."